Lessons Learned Model Checking an Industrial Communications Library

James Ivers

September 2005

Predictable Assembly from Certifiable Components Initiative

Technical Note
CMU/SEI-2005-TN-039

This work is sponsored by the U.S. Department of Defense.

The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Contents

Acknowledgements ..... **vii**

Abstract ..... **ix**

1 Introduction **1**

2 A Model Checking Reasoning Framework **3**

3 Case Study **6**

4 Communications Library Verification **8**
4.1 Document the Design 9
4.2 Formalize Claims 12
4.3 Generate S/R Model 12
4.4 Apply Model Checker 13
4.5 Verification Tuning 14
4.5.1 Applying Manual Abstractions 14
4.5.2 Tuning Model Generation 15
4.5.3 Trying Different Algorithms 15
4.6 Interpret Results 16

5 Results **17**

6 Conclusion **20**

Appendix A Example Fragments **23**

References **27**

List of Figures

Figure 1: Elements of a Reasoning Framework 3

Figure 2: Model Checking Reasoning Framework 4

Figure 3: Sequence Diagram Showing Synchronous Exchange 6

Figure 4: Verification Process 8

Figure 5: Sample Objectbench State Machine 10

Figure 6: Original OCM 11

Figure 7: Sequence Diagram Summarizing a Counterexample 18

Figure 8: Original OIM 24

Figure 9: Final OIM 25

Figure 10: Final OCM 26

List of Tables

Table 1: Problems and Possible Solutions 21

Acknowledgements

This work would not have been possible without the contributions and support of a great group of people. The model checking exercise described in this report was conducted jointly with Natasha Sharygina, whose efforts and experience with the tools used were instrumental in producing successful results. Professor James Browne connected us with HyPerformix, which provided a copy of Objectbench for our research. Likewise, Fei Xie provided a copy of the ObjectCheck toolset. Peter Eriksson and his organization provided us with the communications library to be model checked and with an opportunity to discuss our findings with the engineers responsible for maintaining the library and get their feedback. Sagar Chaki, Linda Northrop, and Kurt Wallnau provided insightful reviews that helped improve the quality of this report.

Abstract

Model checking is a fully automated formal verification technology that can be used to determine whether models of software satisfy behavioral requirements in such areas as safety, reliability, and security. This report explores the packaging of model checking technology in a reasoning framework. The goal of a reasoning framework is to simplify the analysis of software designs by nonexperts. This report describes the application of such a reasoning framework to the design of an industrial communications library and the problems that were found. This report also notes the tasks that were unreasonably complex or time-consuming and concludes with thoughts on techniques that could be used to develop a model checking reasoning framework that better supports use by nonexperts.

1 Introduction

Model checking is a formal verification technique for checking whether a model satisfies specific behavioral requirements [Clarke 99]. Model checking has been successfully applied in numerous domains, such as chip design [Russinoff 98, O'Leary 99], telecommunications [Chaves 92, Chandra 02], and device drivers [Ball 04]. In the software community, model checking is applied both to design artifacts [Hatcliff 03, Holzmann 03] and source code [Ball 01, Henzinger 02] to determine whether software satisfies critical behavioral requirements in such areas as safety, reliability, and security.

Model checkers use a collection of algorithms to verify whether a finite model satisfies user specified claims in *all possible executions*. This exhaustive analysis is an effective means to discover subtle problems, such as race conditions, not easily detected by conventional testing techniques. Additionally, model checking provides diagnostic feedback in the form of counterexamples, which are execution traces that lead to the violation of a claim. This feedback identifies specific circumstances under which problems occur, which in turn helps designers and developers to locate the cause of the failure.

The principle obstacle to successful adoption of model checking for software has been the state space explosion problem. Because the cost of performing exhaustive verification grows exponentially with the size of the models, verification can fail to complete with available resources.[1] However, model checking research has made tremendous progress in recent years, and today's model checking tools include a variety of sophisticated algorithms based on fundamental techniques for coping with state space growth such as

- using more efficient, symbolic representations of models, such as binary decision diagrams (BDDs) [Burch 92]
- pruning the search space based on the equivalence of different paths using partial-order reduction [Godefroid 91, Katz 92, Valmari 91]
- using compositional reasoning techniques to verify portions of the model individually, rather than verifying the explicit composition of all models [Clarke 89]
- using predicate abstraction to construct conservative abstractions that retain the properties of the original models [Graf 97]

Despite such advances in model checking theory, the degree of expertise that is often required to successfully complete verification remains an impediment to widespread adoption of model checking technology by software developers. Ideally, developers should be able to apply model checking to their designs and implementations without learning the intricacies of

[1] Typically, available memory used to store the state space to be searched is the limiting factor, though verification attempts that require tens of hours are also problematic.

model checking theory, the nuances of particular algorithms, or the peculiar notations used by specific tools.

This report explores the packaging of model checking technology in a reasoning framework [Bass 05] that simplifies the analysis of software designs by nonexperts. Software model checking, a term used to refer to the verification of source code, is another important use of model checking technology. In this report, however, we restrict our attention to verifying software designs. The key benefits of verifying design artifacts are that the models are usually smaller and finding problems earlier in the development process results in less costly fixes.

We describe a set of available tools that together effectively form a model checking reasoning framework and their application to the design of an industrial communications library. In addition to presenting the problems discovered by model checking, we note the tasks that were unreasonably complex or time-consuming and conclude with thoughts on techniques that could be used to develop a model checking reasoning framework that better supports use by nonexperts.

2 A Model Checking Reasoning Framework

In addition to functional correctness, software systems typically have to satisfy a variety of quality attribute requirements in such areas as performance, reliability, and security. While conventional testing techniques are used with varying degrees of success to determine whether systems satisfy such requirements, that feedback comes late in the development process, and adequate coverage is difficult to achieve.

Alternatively, specialized theories developed to reason about specific qualities can be applied to predict whether systems will satisfy these requirements prior to implementation or integration. For example, rate monotonic analysis (RMA) is an effective theory for reasoning about system performance given a task structure, execution times, and thread priorities [Klein 93]. Such theories, however, typically require some degree of expertise to apply effectively and use specialized analytic models of the system.

A reasoning framework is a way of packaging an analytic theory and its accompanying models so that nonexperts can reason about a quality attribute [Bass 05]. Figure 1 shows the elements of a reasoning framework. When a user supplies a description of a software architecture that is annotated to satisfy the information requirements of a reasoning framework (e.g., the priority of each thread), the reasoning framework computes a prediction of how the system will behave with respect to some quality attribute (e.g., task latency).

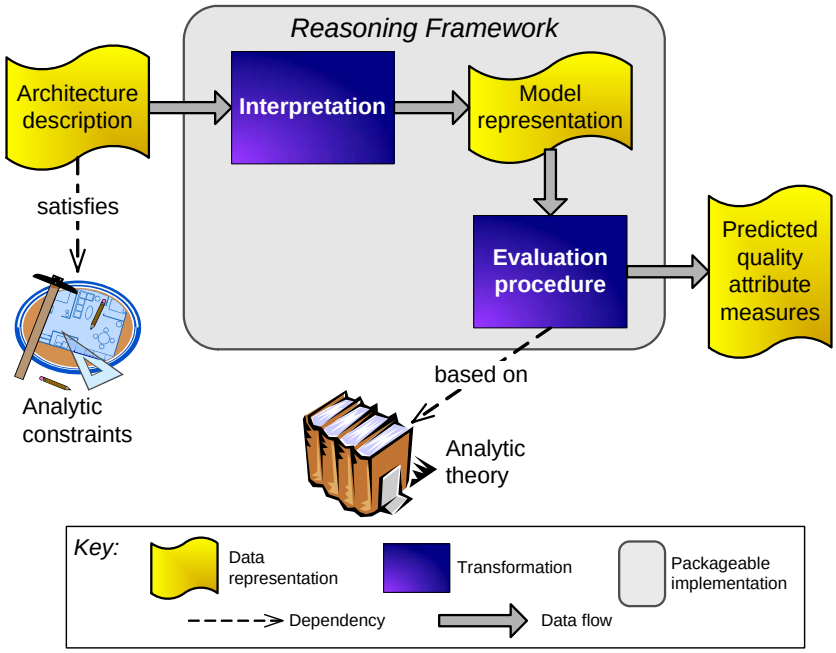

Figure 1: Elements of a Reasoning Framework

Internally, a reasoning framework generates the appropriate model representation for the analytic theory based on the information found in the architecture description. This model is subjected to some evaluation procedure grounded in the analytic theory that computes some set of predicted quality attribute measures. The degree of complexity that is hidden from users depends on the underlying analytic theory and the semantic gap between the analytic models and the software description.

In this exercise, we explored the use of a reasoning framework to package model checking for nonexperts. We chose an existing set of tools for model checking software designs expressed in xUML [Mellor 02]. These tools, shown in Figure 2, closely match the design of a reasoning framework. The principle tools are

- Objectbench: a CASE tool used to generate xUML descriptions. In addition to the class diagrams that typically dominate object-oriented CASE tools, xUML descriptions contain state machines annotated with sufficient behavioral information to generate complete implementations. It is these behavioral descriptions that provide suitable information for model checking [SES 96].
- COSPAN: a commercial model checker implementing a variety of state space reduction techniques that is designed for hardware verification. COSPAN consumes models written in the S/R language [Hardin 96].
- ObjectCheck: tools for generating S/R models from Objectbench descriptions and for generating xUML counterexamples from S/R counterexamples [Xie 02].

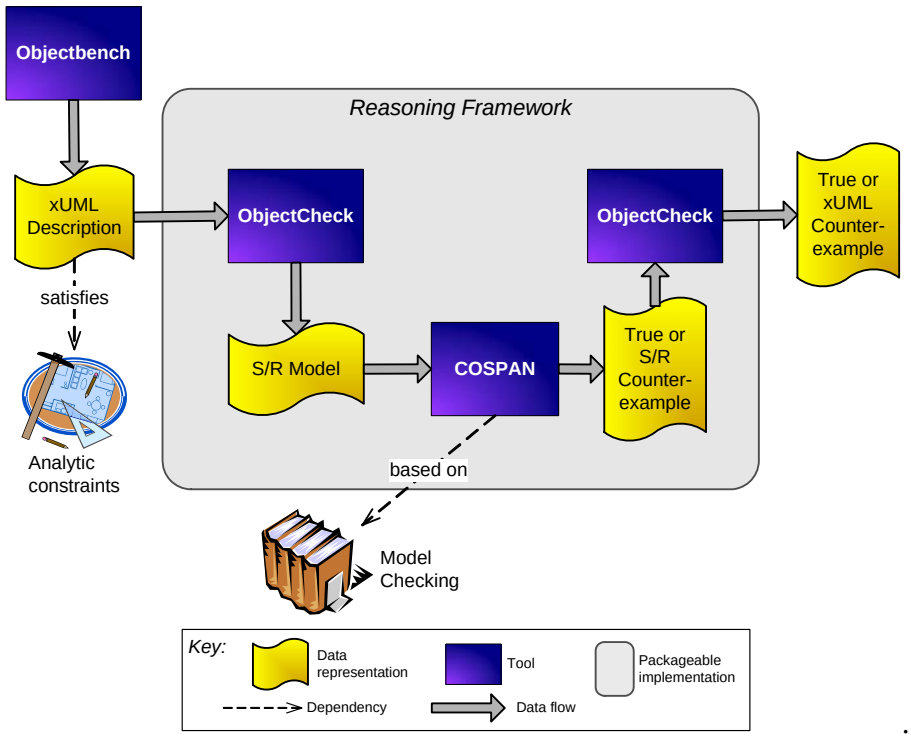

Figure 2: Model Checking Reasoning Framework

While these tools have been used successfully on other problems (e.g., a robot controller [Sharygina 01] and an online ticket sales system [Xie 02]), they were not an ideal match to our goals for a reasoning framework for model checking software designs. At the outset, we were aware of several mismatches:

- Objectbench and xUML are intended to describe object-oriented systems, but we wanted to model check a communications library composed of a collection of C functions. Functions had to be modeled as active objects. While sound, this approach introduced inefficiencies. In particular, each active object has an associated event queue; interactions via function calls, however, are more efficiently verified using event synchronization rather than event queues.
- COSPAN uses a synchronous execution model[2] that best supports hardware models, whereas we would be looking at software, for which an asynchronous execution model is usually more appropriate. ObjectCheck accommodates these differences by generating an S/R model that includes an explicit scheduler process that simulates an asynchronous execution model, though at the cost of increased complexity in the model. This, in turn, translates into greater verification expense.
- The collection of tools was not as opaque to the user as we would envision a reasoning framework to be. Most significantly, for nontrivial problems, users must be expert enough to tweak the models (e.g., by constraining type ranges for variables) and choose verification algorithms in order to coax the verification to completion.

On the whole, this collection of tools proved to be effective and helped us to gain valuable insight into the effectiveness of and difficulties in packaging model checking as a reasoning framework.

[2] In a synchronous execution model, each concurrent process makes progress during each execution step. In an asynchronous execution model, only one process makes progress during each execution step and the behavior of different processes is interleaved over multiple execution steps.

3 Case Study

We applied the collection of tools identified in Section 2 to the design of a communications library that is widely deployed in industrial automation systems. A commercial partner provided us with the source code for this library, which is complex enough to be realistic but readily understandable without significant domain expertise.

The library provides operations for message-based communication among threads. It supports a variety of synchronous and asynchronous forms of communication and includes such realistic but complicating features as

- timeouts on both sender and receiver operations
- shared memory-based message queues implementing a message-based communication style
- three different types of synchronization primitives to coordinate operations invoked by different threads

In particular, we focused on the most complicated portion of the library—those operations used to perform a synchronous message exchange. The sequence diagram in Figure 3 summarizes typical use of the communication library for this case. A sending thread initiates the interaction by sending a message and waiting for the answer (by calling *ipc_sendwait*). A receiving thread requests its next message (*ipc_receive*) and eventually sends a response back to the sending thread (*ipc_answer*).

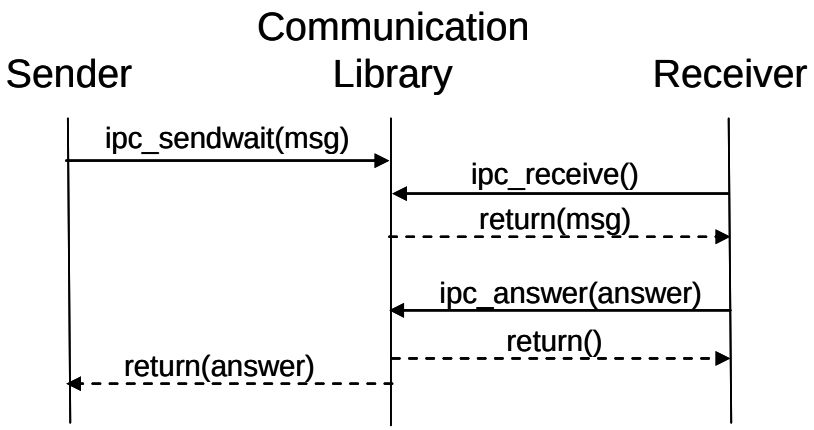

Figure 3: Sequence Diagram Showing Synchronous Exchange

During a quick brainstorming session, we identified 14 claims that (a) this portion of the communications library should satisfy when correctly implemented and (b) are representative of the types of properties that are typically verified using model checking. Some examples include

- If no operations time out, the intended recipient always receives the message sent.
- A sent message is only received by the intended recipient.
- When a sender receives an answer, it is an answer to the sender's most recently sent message.
- A sender (receiver) only blocks when writing to a full message queue (reading from an empty message queue).

In most cases, as demonstrated by extensive testing and field use, the protocol and its implementation behaved as expected. Model checking, however, excels at uncovering unusual or seldom executed paths over which interactions do not behave as expected. In particular, we were interested in whether the combinations of synchronization primitives were used correctly for *all possible executions*, including possible combinations of failures such as timeouts. Any problems discovered might expose a flaw that could cause systems built on top of the communications library to misbehave in seemingly untraceable or irreproducible ways.

4 Communications Library Verification

The nominal process for applying this reasoning framework is relatively simple:

1. Describe the design in Objectbench.
2. Formalize the claims to be verified by model checking.
3. Use ObjectCheck to generate an S/R model.
4. Use COSPAN to verify the S/R model.
5. Interpret the results, generating an xUML counterexample from the S/R counterexample for any claims that do not hold.

The actual experience, however, became somewhat more complicated due to our need to compensate for state space explosion. Figure 4 depicts the resulting process, which shows the additional activities and iterations needed to tune the verification process to a point where model checking became tractable. These steps are elaborated in the following sections (as indicated in parentheses in Figure 4).

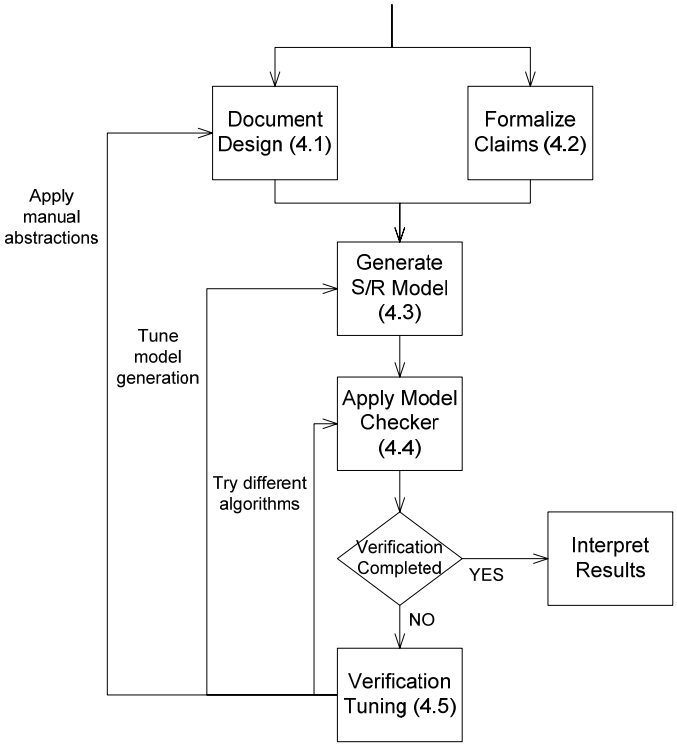

Figure 4: Verification Process

4.1 Document the Design

While design documents were not available to us, we did have access to the source code for the Windows version of the communications library. Creating design models from the source code was a straightforward task once we settled on modeling conventions such as the following:

- Functions are modeled as classes in Objectbench, each of which has its own state machine.
- Function calls are modeled as pairs of events between objects, one representing the call and the other representing the return.
- The shared memory segment used to implement message queues was also represented as a class, and reads and writes were represented using event exchanges.
- Occurrences of timeouts were modeled using nondeterminism; that is, a state machine would nondeterministically decide whether to respond in time or with a timeout event.

For the majority of the state machines, we closely mirrored the control flow found in the corresponding source code (approximately one thousand lines of C code were modeled). For the system calls implementing the various synchronization primitives (critical sections, semaphores, and event objects), we instead relied on documentation from Microsoft. In both cases, various low level details—such as data marshalling, logging, and routine error checking—were omitted to simulate a level of information more indicative of design models.

An example of the type of state machines that were created is shown in Figure 5. The states and transitions reflect the control flow of the corresponding function, and states are annotated with action statements that combine C code with `Generate` statements for sending an event to another state machine.

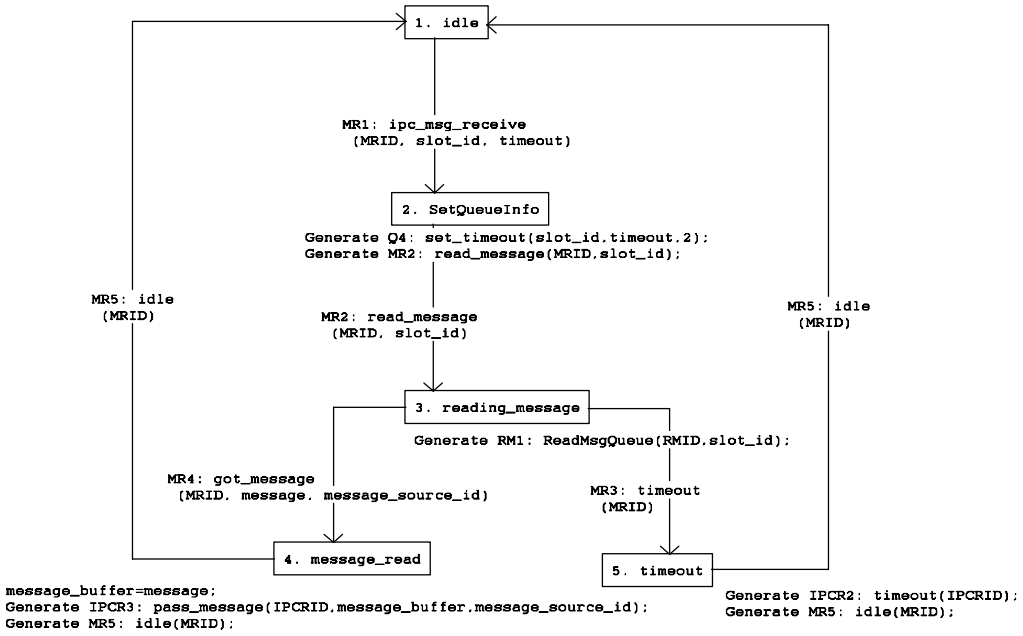

Figure 5: Sample Objectbench State Machine

In fact, the syntax used in Objectbench to denote sending events from one state machine to another became problematic. It introduced a tight binding between state machines, particularly with regard to our convention of representing function returns by events sent back to the caller, which made the state machines less reusable. Moreover, it forced us to use awkward modeling conventions in cases in which multiple functions could call the same function. In these cases, we use a parameter of the call event to determine which event to generate to represent the function return.

In addition to modeling the functions from the communication library and certain system calls, we had to create models for senders and receivers to simulate use of the library. The sender's behavior was to complete initialization and enter a loop in which it would nondeterministically pick a receiver, send the receiver a message, and wait for the answer. The receiver's behavior was even simpler; after completing initialization, it entered a loop in which it would attempt to receive its next message and then send an answer. We used integers to simulate messages and answers; further, receivers were designed to answer with the same integer message they received, allowing us to easily match messages and answers.

We also had to decide how many senders and receivers to include in our verification scenario. We created general-purpose senders and receivers so that we could instantiate various numbers of each and ultimately move to a peer-based model in which any process could send or receive. For each sender and receiver instantiated, corresponding instances had to be created of the models of all functions called by those processes. For example, senders call IPCSendWait, so one instance of IPCSendWait must be created for each instance of Sender. We began with one sender and two receivers, and the complete original Objectbench model contained a total of 46 instances.

Figure 6 shows a screen shot of the original Object Communication Model (OCM) for the design of the communication library. Each bubble in the OCM represents a class, and each labeled line represents an event that can be sent from one class to another. While the model does not indicate the number of instances of each class, the degree of connectedness should give the reader an impression of the number of interactions among elements of the library.

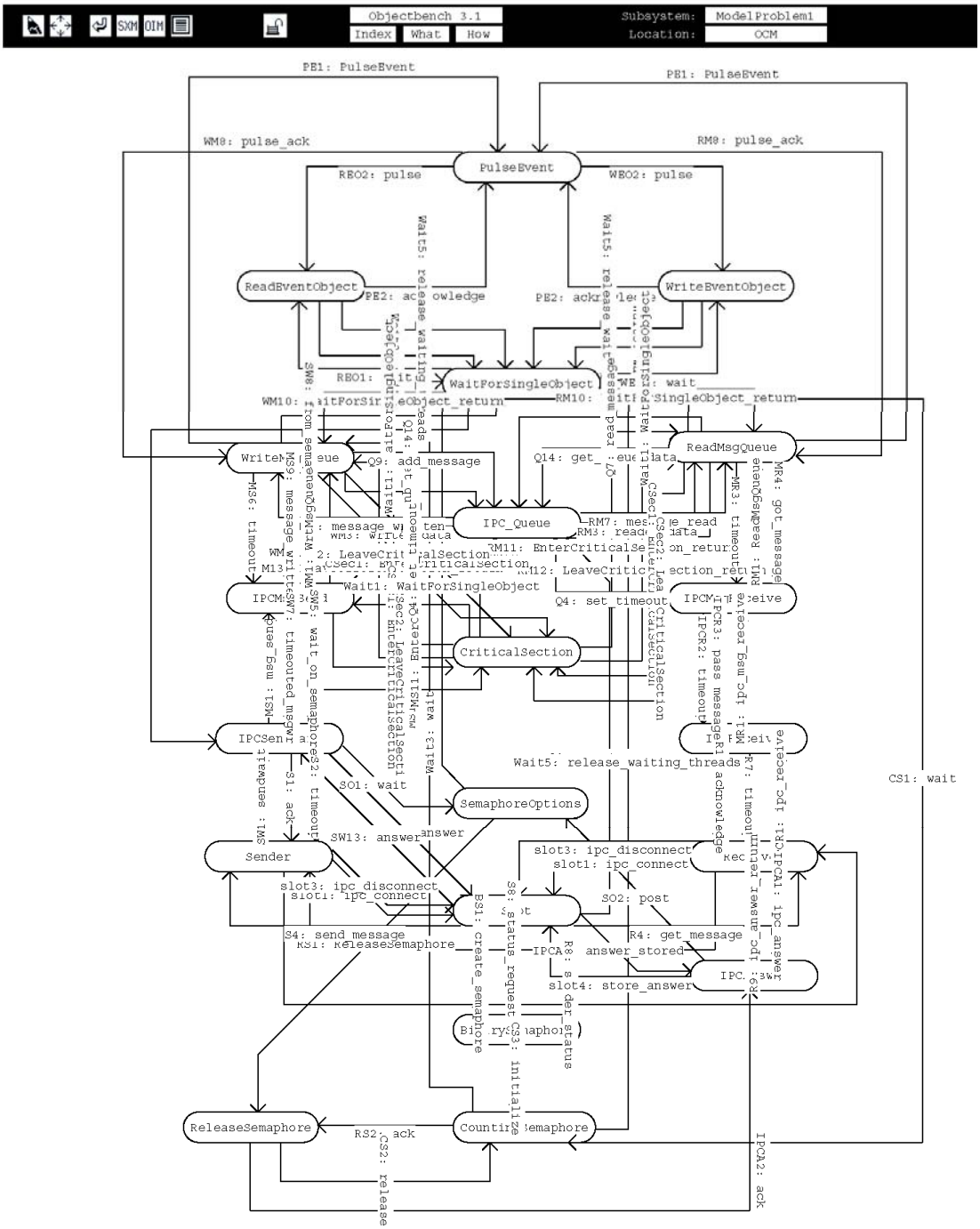

Figure 6: Original OCM

4.2 Formalize Claims

When brainstorming claims, we simply recorded each claim in prose. To model check a claim, though, it must be formalized in a language understood by the model checker. In the case of COSPAN, we recorded each claim as a linear temporal logic (LTL) formula [Manna 92] that referred to variables from the Objectbench models. ObjectCheck then translated these claims to refer to the corresponding variables in the generated S/R models.

For example, consider this prose claim: a sender only blocks when attempting to write to a full message queue. The natural expression of this claim refers to both events (invoking the system call `WaitForSingleObject` to block) and state information (the current size and capacity of the destination message queue). However, most variants of LTL and the specific variant that ObjectCheck translates to S/R only permit references to state. Consequently, we had to annotate the Objectbench model with additional flag variables used to denote interesting points in the execution—like calling `WaitForSingleObject`. Formalized in LTL, we express this claim as

```
G (WRITEMSGQUEUE(1).FLAG==1 && WRITEMSGQUEUE(1).QUEUE_ID==1
    --> IPC_QUEUE(2).NUMMESSAGES == 3)
```

A prose translation of this "it is always true that when `FLAG == 1` and `QUEUE_ID == 1`, `NUMMESSAGES` must be `3`." That is, for the first sender instance, whenever it blocks (`FLAG == 1`), the destination queue must be full (the current value of `NUMMESSAGES` equals the size of the queue—3 in this case). `FLAG` is one of the variables we had to introduce into the Objectbench model for the sole purpose of writing claims to be model checked; blocking is actually represented by generating a `WaitForSingleObject` event, which corresponds to invoking a system call to wait on a synchronization condition.

The inability to cleanly refer to the occurrence of events caused us to add such additional variables, which further exacerbated the state space explosion problem, reduced clarity, and limited the ability to review that each formalized claim correctly matched the intent.

4.3 Generate S/R Model

This was by far the simplest step in our process. As long as our Objectbench models conformed to the constraints imposed by ObjectCheck, this was a completely automated step. One notable restriction, however, was the set of allowable datatypes—only integers could be used in the Objectbench models. This meant modeling Booleans, for example, as a more expensive datatype.

As part of generating an S/R model, ObjectCheck sets a bound on the range of values each integer variable could have (a necessary step to define a finite-state machine). The default range was `-1024..1023`. This range was much larger than necessary for all variables in our model of the communications library (even for the integer representing messages and

answers, because we designed the sender to reset this value to zero after N iterations) and was something we adjusted during verification tuning (see Section 4.5 for more information).

Automation was invaluable for this step. The size and complexity of the generated S/R models were well beyond what we could have reasonably produced manually, and the semantic gap between the models and designs would have significantly complicated any peer review process. As an example, the textual representation of the final version of our Objectbench models is a 1,698-line file; the corresponding S/R model is a 7,710-line file.

4.4 Apply Model Checker

Naïve application of the model checker consists of simply executing COSPAN over the generated S/R model. On applying COSPAN to our initial model of the communications library, it reported an estimated state space of 2.35×10^{1932} states—well beyond what was tractable. While COSPAN can be applied in a more sophisticated manner by using command line options to apply various state space reduction algorithms—such as using a symbolic representation (BDDs), partial order reduction, and bounded model checking—it was obvious that we first needed to reduce the size of the problem before these optimizations would be enough to complete verification.

After examining how ObjectCheck generates S/R models, we noted a number of factors contributing to the large state space:

- As already noted, the default range for integer values was much larger than necessary for the variables in the model, usually by three orders of magnitude.
- As noted in Section 2, the models ObjectCheck generates are not ideal for the kind of software we were analyzing. For example, given that we were modeling function calls without their own threads of control, most state machines did not need the accompanying event queue generated by ObjectCheck. Each such event queue constituted an additional process, and state space growth is exponential in the number of processes.
- The Objectbench model and corresponding S/R model contained many unnecessary variables, particularly the referential attributes used to identify instances to which a state machine could send events. When creating the Objectbench models, we adopted a convention for assigning instances such that there would be one instance of each state machine representing a function that could be called by a thread of control (specifically, the senders and receivers) and the ID attribute of each such instance would have the same value as the ID of the thread of control. Consequently, all referential attributes in most instances would always have the same value, and only one copy really needed to be included.
- The models we created were more general than necessary for a verification scenario involving strict senders and receivers. We created the models with the hope of generalizing the verification scenario by modeling a collection of peers that could send

and receive. A consequence of this generalization was that more instances were present in the model than were necessary.

With these observations in mind, we began tuning the models to ameliorate state space growth, as discussed in the next section.

4.5 Verification Tuning

As shown in Figure 4, the various modifications we applied fell into three categories:

1. applying manual abstractions
2. tuning model generation
3. trying different algorithms

The following sections describe some of the modifications we performed in each category. We used an incremental approach, going for big reductions first and iterating until we produced a state space small enough to complete verification. The result of this tuning was to reduce the state space estimates from 2.35×10^{1932} states to 6.07×10^{233} states, only a small fraction of which were searched in locating the problems we found (details on problems found are reported in Section 5).

4.5.1 Applying Manual Abstractions

When applying manual abstractions, the goal is to make the model smaller without losing any information relevant to the claims of interest. The biggest difference can be made by eliminating unnecessary state machines (recall that the size of the state space is exponential in the number of processes and that each state machine results in two processes due to the addition of the implicit event queue). Over a series of changes, we were able to eliminate 26 of the 46 object instances in our original Objectbench model by

- reducing the number of receivers from two to one, which eliminated all 15 object instances in that receiver's thread of control
- removing the flexibility for each sender or receiver to become a peer and engage in both sending and receiving behavior
- removing a critical section that was only used to coordinate writes among senders, which was unnecessary with only one sender included
- removing a pair of classes used to model delegations that existed in the code (i.e., one function that only calls another using specific parameters) by inlining the delegation in the state machine modeling the original calling function

Eliminating more than half of the object instances was one of the two largest sources of improvement in our manual state space reduction efforts.

We also eliminated a number of variables from the Objectbench models. The largest gain here was in the removal of the many unnecessary referential attributes mentioned in Section 4.4.

Additionally, we were able to eliminate about 20 variables by reducing the size of the message queues used in the communications library (not to be confused with the implicit event queues generated by ObjectCheck) and synchronization functions. Many of these reductions were only safe given knowledge of the usage scenario (the number of senders and receivers and the maximum queue utilization) for which we would be verifying the behavior of the communications library.

4.5.2 Tuning Model Generation

While we could not disable the generation of the implicit event queues by ObjectCheck in S/R models, we were able to tune the range of values of each variable. Bounding variables is a commonly used technique for managing state space growth but must be performed carefully. Bounding an integer variable to a small, finite range such as `0..5` may dramatically reduce the state space size, but it can also introduce problems in the model. First, should the range be too small to permit problematic behavior to arise, model checking will produce a false sense of confidence. Second, the ranges of variables must be coordinated. If `x` has a range of `0..5` and `y` has a range of `0..5`, then `z` should usually have a range of `0..10` if the statement `z = x + y;` appears in the model. Determining a consistent set of ranges can be, therefore, a tedious task.

For the communications library, the basic process was to trace each variable to the source of its values (often by tracing back through several function calls or events exchanged among state machines) and determining or constraining the maximum set of possible values given the number of object instances being created in our scenario. The default range for all variables was `-1024..1023`. In the final version, this range was three orders of magnitude smaller for all variables, with some having ranges as small as `0..1`. This form of tuning was one of the two largest sources of improvement in our manual state space reduction efforts.

4.5.3 Trying Different Algorithms

At various points in the process of manually reducing the size of the state space by using the techniques described in Sections 4.5.1 and 4.5.2, we would attempt to model check the resulting S/R models. Until we were able to get the state space estimate down to the 10^{300} range, however, we were unable to get any results within a reasonable period of time[3] or without exhausting available memory.

COSPAN provides a variety of state space reduction algorithms that can be applied by supplying the appropriate command line arguments. Over the course of various experiments and iterations, we used a variety of these algorithms to attempt to complete verification, including

[3] Our subjective measure of *reasonable* was an overnight job on cluster machines. Sole access to these machines was atypical, however, and better results or more timely feedback should be expected given better computing resources.

- symbolic representation with BDDs to represent models very efficiently in memory
- bounded model checking for limiting the search depth of any particular path[4]
- partial-order reduction, which avoids searching interleavings that are equivalent with respect to claims of interest

Of the model checking attempts we were able to complete with available resources and patience, all used an explicit (not symbolic) state space search using partial-order reduction. That's not to say that other techniques aren't helpful or that others couldn't have been tried; but with our general goal of minimal manual intervention, this approach turned out to be what worked.

4.6 Interpret Results

Interpreting model checking results involves two steps with this reasoning framework. First, all counterexamples generated by COSPAN are execution traces of the S/R model, something not suitable for user consumption. ObjectCheck automates the production of a usable version by generating corresponding execution traces of the Objectbench models from the S/R counterexamples. This form allows a user to relate the trace leading to a problem to the models he or she created in the first place.

The more difficult step is examining the Objectbench counterexamples and understanding the cause(s) of the indicated problems. Counterexamples can be quite long, particularly when problems only manifest after a significant period of execution necessary to put the system into particular states, such as error recovery modes. One particular counterexample we examined uses 183 steps to demonstrate a problem.[5]

In such circumstances, we found it helpful to sketch a sequence diagram abstracting the counterexample as we read. By manually producing a version in which perhaps dozens of low-level execution steps are replaced by abstractions like "read message and send answer," we created a more manageable set of sequenced steps separated by interleavings (context switches). See Figure 7 in the following section for an example.

[4] The effectiveness of bounded model checking often seems counterintuitive. However, when model checking finds problems, it is usually after searching a fraction of the state space; only when claims hold must the full state space be searched.

[5] For reference, the S/R version of this counterexample is a 376-line file.

5 Results

Despite various inconveniences encountered during our application of the model checking reasoning framework to the design of the communications library, it was a valuable exercise. In addition to identifying problems and areas for improvement for future model checking reasoning frameworks, we discovered several problems with the design of the communications library.

Specifically, the following claims did not hold:

1. When a sender receives an answer, it is an answer to the sender's most recently sent message.
2. A sender only blocks when writing to a full message queue.
3. Answers are not delivered to an inactive slot.

These problems varied in importance, with the first being most significant in that a system could take inappropriate action based on incorrect responses, the second potentially degrading system performance, and the third resulting in generation of the wrong error message.

Regarding the first problem, Figure 7 illustrates a summary of an execution of the communications library, focusing on the use of a semaphore that leads to the problem. This result was a product of the final Objectbench model, whose corresponding S/R model contained an estimated 6.07×10^{233} states. The problem, consisting of 183 steps though the 20 object instances from the Objectbench model, was found after searching only 19,759 states. Total verification time, however, was approximately 2.5 hours, the majority of which was needed to construct the state space prior to starting the search.

To summarize the problem, it is possible for a sender and receiver communicating synchronously to get "out-of-sync" if the proper sequence of timeouts and context switches occurs. The problem arises when the sender sends a message and times out before an answer is received. When the sender sends its next message, the receiver might finally answer the first message at a point when the sender is expecting an answer to the second message.

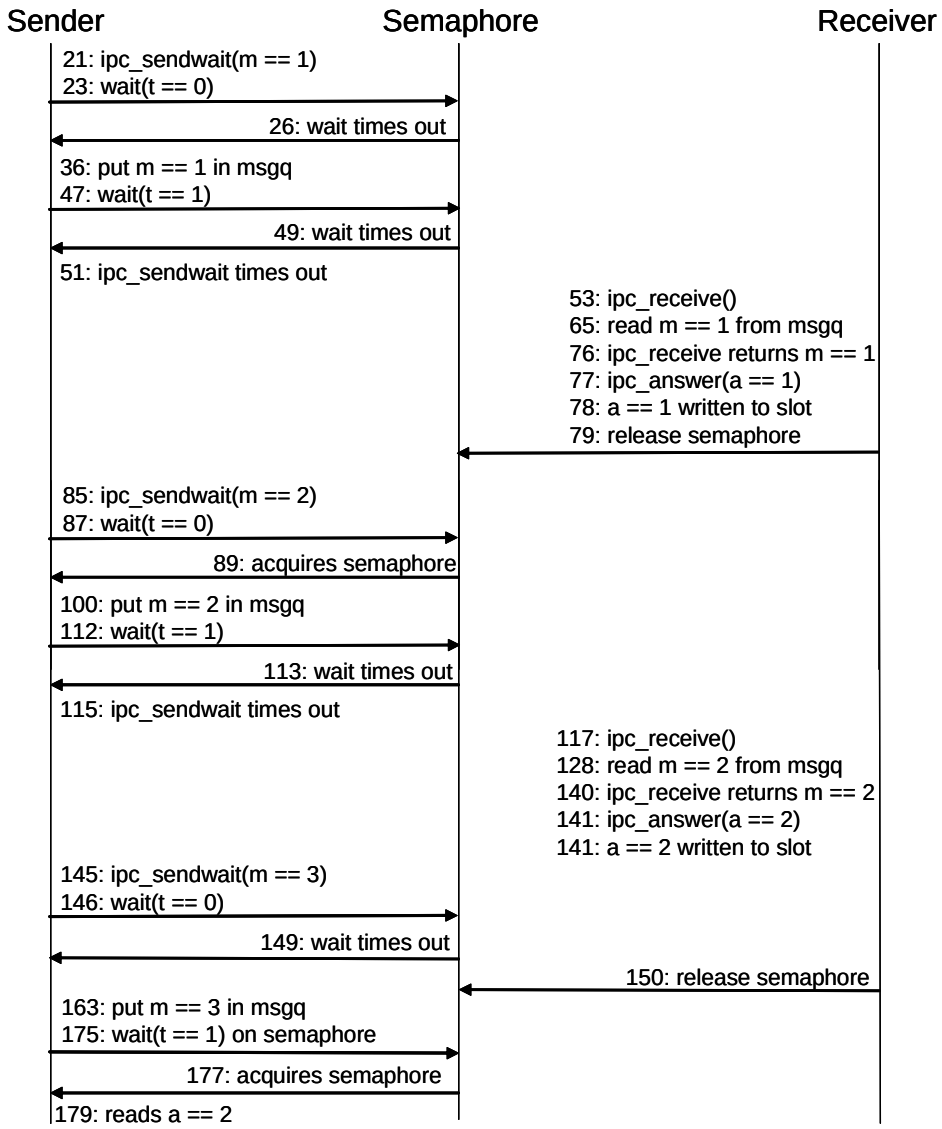

Figure 7: Sequence Diagram Summarizing a Counterexample

Figure 7 shows one way in which we used the xUML counterexample—to produce a summary for communicating with the engineers responsible for maintaining the communications library. The information found in the counterexample was also sufficient to find the relevant bits of source code, confirm the existence of the problem by code inspection, and to help us to write a test program demonstrating the problem.

While model checking was successfully used to discover the three cited problems, we had considerably more difficulty using model checking to show the absence of problems. Verification attempts on other claims typically exhausted available memory or were aborted after lengthy executions (some after more than 12 hours). This does not diminish the importance of the findings, however. According to one engineer, the problem of a sender receiving the wrong answer had existed in the communications library for more than seven

years before it was identified.[6] It is an exemplar of the kinds of subtle problems caused by race conditions that are notoriously difficult to locate or reproduce via testing but routinely uncovered by the exhaustive searches used in model checking.

[6] The engineers had, in fact, found the problem independently of our model checking effort. We were analyzing an older version of the library in which the problem still existed.

6 Conclusion

Model checking the communications library was a success in two respects. First, as is typical of model checking exercises, we were able to find and verify the existence of design and implementation problems, one of which was quite serious (though seldom encountered). This process provided valuable feedback to the engineers maintaining the communications library and helped them understand the effectiveness of model checking in discovering these kinds of subtle problems.

Second, and more important from a research perspective, we compared the application of the collection of tools to our ideal expectations of a model checking reasoning framework and identified tasks that are unreasonably complex or time consuming for application by nonexperts or when dealing with a larger code base. This comparison helped fuel our development of the ComFoRT model checking reasoning framework [Ivers 04].

Our goals for ComFoRT are twofold:

1. Support the verification of industrial-scale problems by incorporating state-of-the-art model checking algorithms and associated techniques for generating efficient models from software designs.
2. Support verification by nonexperts by eliminating the need for the types of manual intervention described in Sections 4.2 and 4.5 of this report.

Table 1 summarizes the most significant problems we encountered while model checking the communications library and techniques we are investigating to address them in ComFoRT.

Problem	Possible Solution
Objectbench models are tightly coupled due to a need to identify the destination of each event; this characteristic complicates the model logic dealing with generating responses and limits model reuse.	Architecture description languages and composition languages provide a better model for isolating the behavior of one component from that of another. Binding decisions can be deferred until components are instantiated and wired together, at which point the model of each instance can be tailored as needed.
The language used to formalize claims does not permit references to events, forcing the user to instrument Objectbench models with additional information that has nothing to do with the design or implementation of the software.	A state/event temporal logic [Chaki 04] can be used that allows claims to reference both states and events, as was needed for many of the claims we identified for the communications library.
The mismatches between the toolset and the design of the communications library resulted in the generation of an overly complex (large) model.	Model generation should use conventions that more closely match the semantics of the designs being modeled. A suitable set of conventions should be easily identifiable for a constrained domain, such as systems built using a specific component technology.
The size of the generated model was too large for the model checking tool to manage automatically; forcing the user to try various manual state space reduction techniques, some of which are tedious or error-prone.	Many of the manual techniques we used (e.g., limiting the number of receivers) were point solutions and have no systematic solution beyond generic state space reduction algorithms. The problem of variable ranges, however, can be addressed by using techniques based on predicate abstraction. These techniques use predicates over variable values rather then enumerating them over a finite range, which dramatically reduces verification complexity in addition to eliminating the need for user calculations.

Table 1: Problems and Possible Solutions

The current version of ComFoRT implements, to differing degrees of completeness, each of the solutions identified in Table 1 and has been successfully applied to a subset of the communications library with substantially less effort than reported in Section 4. We are currently refining these solutions and researching other alternatives to improve the scalability and applicability to industrial problems of ComFoRT while retaining its usability by nonexperts.

Appendix A Example Fragments

The diagrams in this appendix are shown to give the reader an impression of the degree to which the structure of the Objectbench models changed during the process of verification tuning (see Section 4.5 for more information about specific changes).

Figure 8 shows the original (pre-tuning) Object Information Model (OIM). This diagram shows a box for each class in the model along with their attributes. Each line represents a relationship between two classes, indicating that one can identify the other. Each line is further represented by a reference attribute (e.g., REQ_ID in PulseEvent) in which the run-time value of an association is stored.

Figure 9 shows the final (post-tuning) version of the OIM. The final version is much smaller due to the elimination of two classes, of many associations that are redundant given modeling conventions, and of many attributes.

Figure 10 shows the final version of the OCM. Again, the final version is simplified in part due to the elimination of two classes. It is also simplified by the elimination of some functionality (e.g., the critical section used to coordinate write attempts), which eliminated some paths of communication. For comparison, see the original OCM shown in Figure 6.

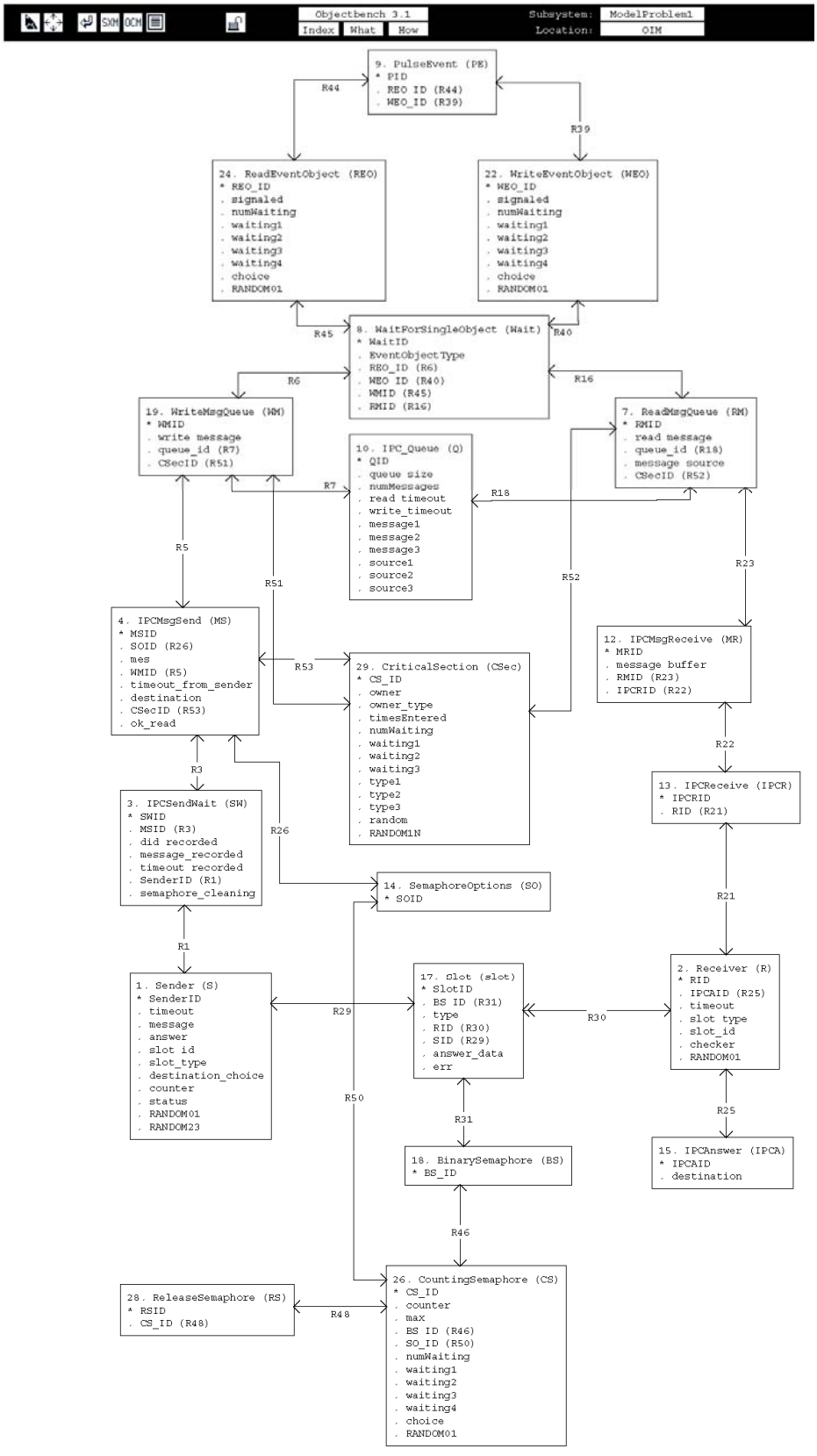

Figure 8: Original OIM

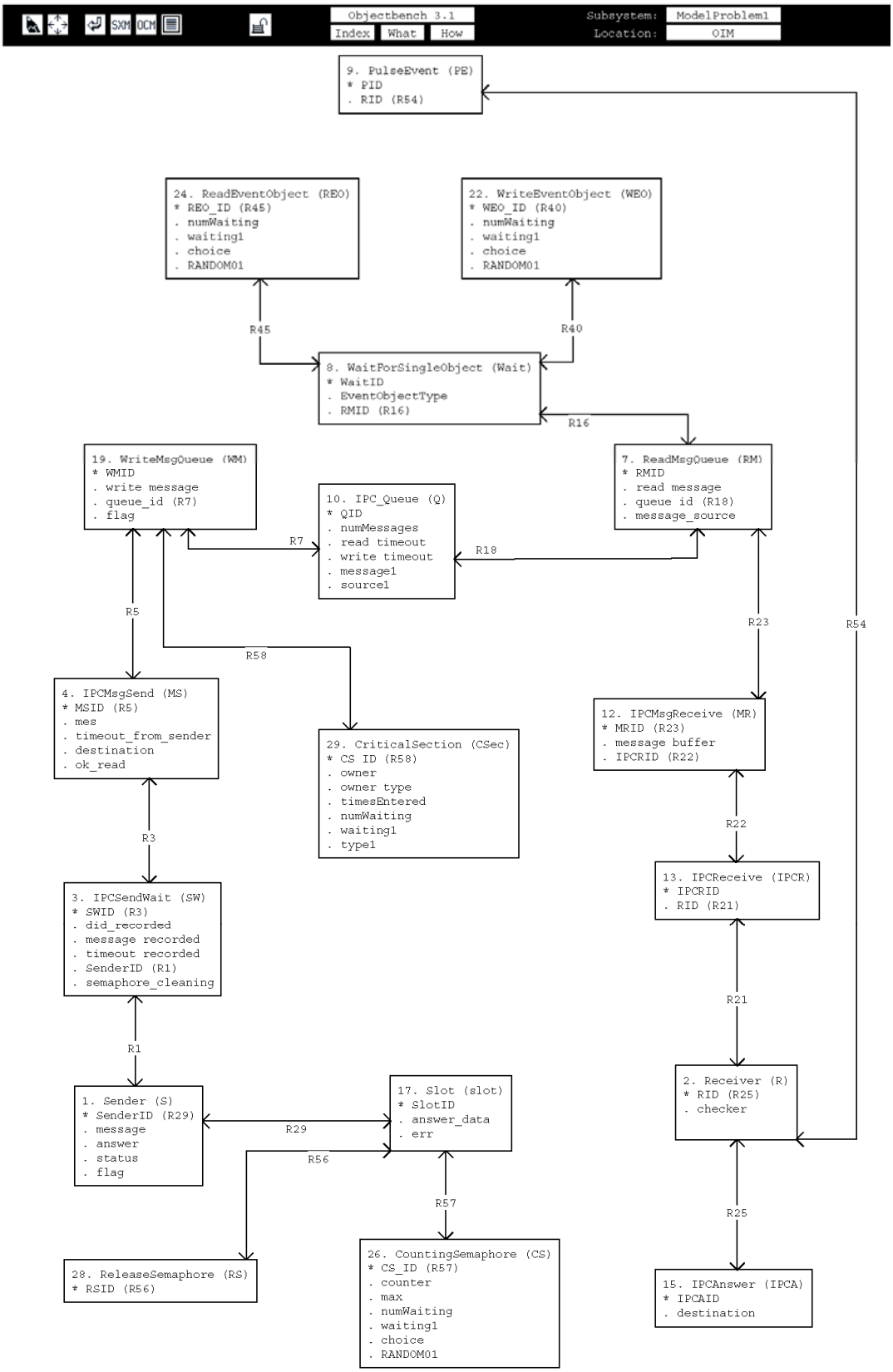

Figure 9: Final OIM

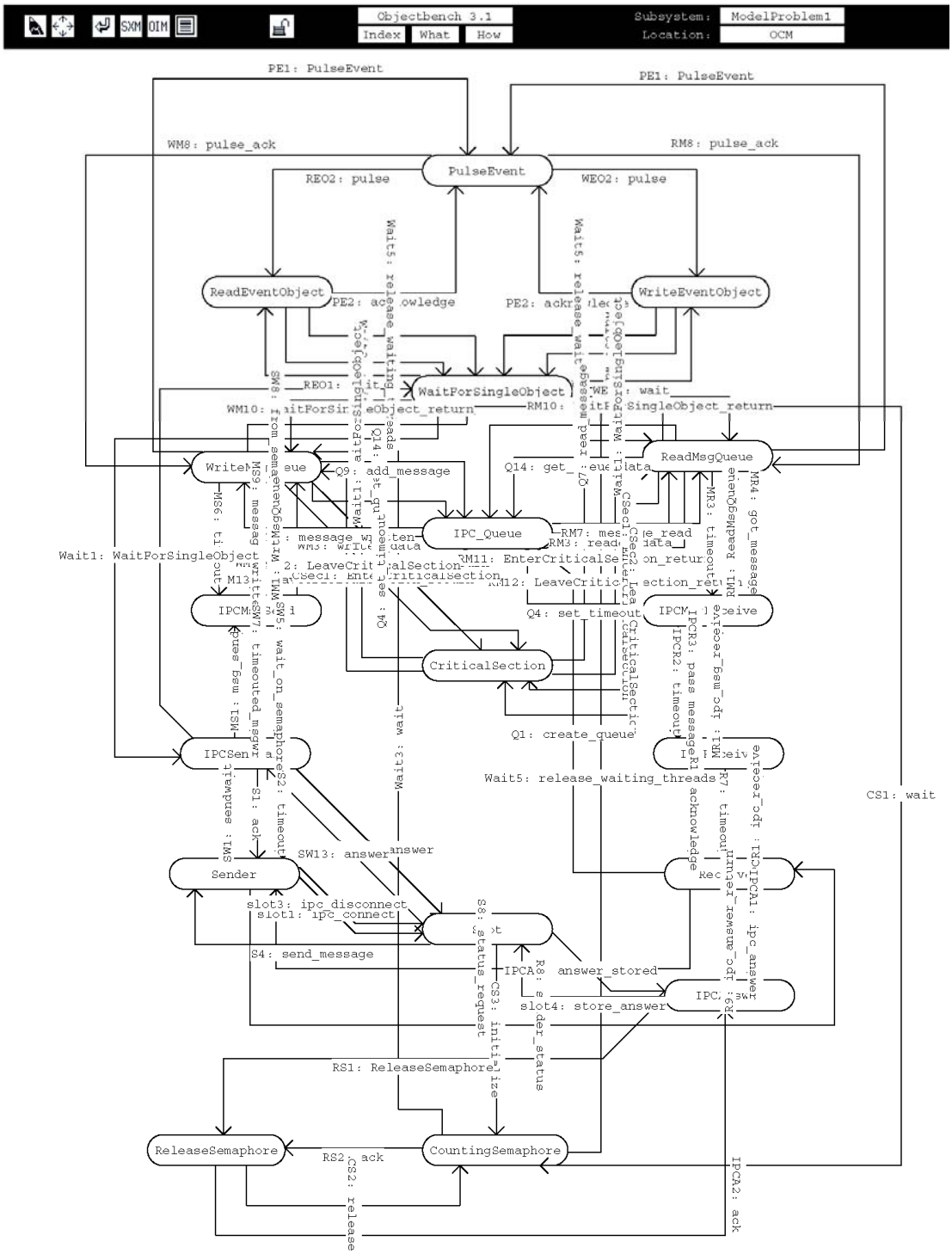

Figure 10: Final OCM

References

URLs are valid as of the publication date of this document.

[Ball 01] Ball, T. & Rajamani, S. “Automatically Validating Temporal Safety Properties of Interfaces,” 103-122. *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN ‘01)* (Lecture Notes in Computer Science [LNCS], volume 2057). Toronto, Canada, May 19-20, 2001. Berlin, Germany: Springer-Verlag, 2001.

[Ball 04] Ball, T.; Cook, B.; Levin, V.; & Rajamani, S. “SLAM and Static Driver Verifier: Technology Transfer of Formal Methods Inside Microsoft,” 1-20. *Integrated Formal Methods: 4th International Conference (IFM 2004)*. Canterbury, Kent, UK, April 4-7, 2004. New York, NY: Springer-Verlag, 2004.

[Bass 05] Bass, L.; Ivers, J.; Klein, M.; & Merson, P. *Reasoning Frameworks* (CMU/SEI-2005-TR-007). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005. http://www.sei.cmu.edu/publications/documents/05.reports/05tr007/05tr007.html

[Burch 92] Burch, J. R.; Clarke, E. M.; McMillan, K. L.; Dill, D. L.; & Hwang, L. J. “Symbolic Model Checking: 10^{20} States and Beyond.” *Information and Computation 98*, 2 (June 1992): 142-170.

[Chaki 04] Chaki, S.; Clarke, E.; Ouaknine, J.; Sharygina, N.; & Sinha, N. “State/Event-Based Software Model Checking,” 128-147. *Integrated Formal Methods: The 4th International Conference (IFM 2004)* (Lecture Notes in Computer Science [LNCS], volume 2999). Canterbury, Kent, UK, April 4-7, 2004. Berlin, Germany: Springer-Verlag, 2004.

[Chandra 02] Chandra, S.; Godefroid, P.; & Palm, C. “Software Model Checking in Practice: An Industrial Case Study,” 431-441. *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*. Orlando, FL, May 19-25, 2002. New York, NY: Association for Computing Machinery (ACM), 2002.

[Chaves 92] Chaves, J. “Formal Methods at AT&T: An Industrial Usage Report,” 83-90. *Proceedings of Formal Description Techniques IV*. Sydney, Australia, November 19-22, 1991. Amsterdam, Netherlands: North-Holland, 1992 (ISBN 0-444-89402-0).

[Clarke 89] Clarke, E.; Long, D.; & McMillan, K. “Compositional Model Checking,” 353-362. *Proceedings of the 4th Annual IEEE Symposium on Logic in Computer Science (LICS ‘89)*. Asilomar, CA, June 5-8, 1989. Piscataway, NJ: IEEE Computer Society Press, 1989 (ISBN 0-8186-1954-6).

[Clarke 99] Clarke, E.; Grumberg, O.; & Peled, D. *Model Checking*. Cambridge, MA: MIT Press, 1999.

[Godefroid 91] Godefroid, P. “Using Partial Orders to Improve Automatic Verification Methods,” 321-339. *Proceedings of the 2nd International Workshop on Computer Aided Verification (CAV 1990)*. (Lecture Notes in Computer Science [LNCS], volume 531). New Brunswick, NJ, June 18-21, 1990. Berlin, Germany: Springer-Verlag, 1991.

[Graf 97] Graf, S. & Saïdi, H. “Construction of Abstract State Graphs with PVS,” 72-83. *Proceedings of the Computer Aided Verification 9th International Conference (CAV ‘97)* (Lecture Notes in Computer Science [LNCS], volume 1254). Haifa, Israel, June 22-25, 1997. Berlin, Germany: Springer-Verlag, 1997.

[Hardin 96] Hardin, R.; Har’El, Z.; & Kurshan, R. P. “COSPAN,” 423-427. *Proceedings of Computer Aided Verification 8th International Conference (CAV 1996)* (Lecture Notes in Computer Science [LNCS], volume 1102). New Brunswick, NJ, July 31-August 3, 1996. Berlin, Germany: Springer-Verlag, 1996.

[Hatcliff 03] Hatcliff, J.; Deng, X.; Dwyer, M. B.; Jung, G.; & Ranganath, V. P. “Cadena: An Integrated Development, Analysis, and Verification Environment for Component-Based Systems,” 160-173. *Proceedings of 25th International Conference on Software Engineering (ICSE 2003)*. Portland, OR, May 3-10, 2003. Piscataway, NJ: IEEE Computer Society Press, 2003.

[Henzinger 02] Henzinger, T. A.; Jhala, R.; Majumdar, R.; & Sutre, G. “Lazy Abstraction,” 58-70. *Proceedings of 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’02)* (SIGPLAN Notices, volume 37[1]). New York, NY: Association for

Computing Machinery (ACM), 2002.

[Holzmann 03] Holzmann, G. J. *The SPIN Model Checker: Primer and Reference Manual*. Boston, MA: Addison-Wesley, 2003.

[Ivers 04] Ivers, J. & Sharygina, N. *Overview of ComFoRT: A Model Checking Reasoning Framework* (CMU/SEI-2004-TN-018). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2004. http://www.sei.cmu.edu/publications/documents/04.reports/04tn018.html

[Katz 92] Katz, S. & Peled, D. "Verification of Distributed Programs Using Representative Interleaving Sequences." *Distributed Computing 6*, 2 (1992): 107-120.

[Klein 93] Klein, M. H.; Ralya, T.; Pollak, B.; Obenza, R.; & Harbour, M. G. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems.* Boston, MA: Kluwer Academic Publishers, 1993.

[Manna 92] Manna, Z. & Pnueli, A. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* New York, NY: Springer-Verlag, 1992.

[Mellor 02] Mellor, S. & Balcer, M. J. *Executable UML: A Foundation for Model-Driven Architecture*. Boston, MA: Addison-Wesley, 2002.

[O'Leary 99] O'Leary, J.; Zhao, X.; Gerth, R.; & Seger, C. H. "Formally Verifying IEEE Compliance of Floating-Point Hardware." *Intel Technology Journal 3*, 1 (February 1999).

[Russinoff 98] Russinoff, D. "A Mechanically Checked Proof of IEEE Compliance of the Floating-Point Multiplication, Division, and Square Root Algorithms of the AMD-K7* Processor." *London Mathematical Society Journal of Computation and Mathematics 1* (December 1998): 148-200.

[SES 96] Scientific & Engineering Solutions. *SES/objectbench User's Manual*. Annapolis Junction, MD: SES, Inc., 1996.

[Sharygina 01] Sharygina, N.; Kurshan, R. P.; & Browne, J. C. "A Formal Object-oriented Analysis for Software Reliability," 318-332. *Proceedings of the 4th International Conference on FASE* 2001 (Lecture Notes in Computer Science [LNCS], volume 2029). Genova, Italy, April

2-6, 2001. Berlin, Germany: Springer-Verlag, 2001.

[Valmari 91] Valmari, A. “Stubborn Set for Reduced State Space Generation,” 1-22. *Proceedings of the 10th International Conference on Application and Theory of Petri Nets, Volume 2*. (Lecture Notes in Computer Science [LCNS], volume 483). Bonn, Germany, June 1989. Berlin, Germany: Springer-Verlag, 1991.

[Wallnau 03] Wallnau, K. *Volume III: A Technology for Predictable Assembly from Certifiable Components* (CMU/SEI-2003-TR-009, ADA413574). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. http://www.sei.cmu.edu/publications/documents/03.reports/03tr009.html

[Xie 02] Xie, F.; Browne, J. C.; & Levin, V. “ObjectCheck: Model Checking Tool for Model Checking Executable Object-Oriented Software Designs,” 64-79. *Proceedings of FASE 2002* (Lecture Notes in Computer Science [LCNS], volume 489). Grenoble, France, April 8-12, 2002. Berlin, Germany: Springer-Verlag, 2002.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	September 2005	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Lessons Learned Model Checking an Industrial Communications Library	FA8721-05-C-0003

6. AUTHOR(S)

James Ivers

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2005-TN-039

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Model checking is a fully automated formal verification technology that can be used to determine whether models of software satisfy behavioral requirements in such areas as safety, reliability, and security. This report explores the packaging of model checking technology in a reasoning framework. The goal of a reasoning framework is to simplify the analysis of software designs by nonexperts. This report describes the application of such a reasoning framework to the design of an industrial communications library and the problems that were found. This report also notes the tasks that were unreasonably complex or time-consuming and concludes with thoughts on techniques that could be used to develop a model checking reasoning framework that better supports use by nonexperts.

14. SUBJECT TERMS	15. NUMBER OF PAGES
model checking, reasoning framework, software design analysis	42

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL